Pandas Series (`pd.Series`)

At the beginning of most EDAs, you will see the following two lines which import the pandas and numpy libraries. The general convention is to use `pd` to represent the pandas library and `np` to represent the numpy library

In [1]:
```
import pandas as pd
import numpy as np
```

Pandas Series

The building block of pandas is a `pd.Series` object, which is an indexed, sequential list of data. We will be using crypto market caps below to set the example:

In [2]:
```
market_caps = pd.Series([954.7, 514.4, 95.8, 76.3, 57.9, 45.7, 41.0, 38.7, 28.8, 25.8]
```

In [3]:
```
market_caps
```

Out[3]:
```
0    954.7
1    514.4
2     95.8
3     76.3
4     57.9
5     45.7
6     41.0
7     38.7
8     28.8
9     25.8
dtype: float64
```

To make this data clearer, we can add a `name` to the Series:

In [4]:
```
market_caps.name = 'Market caps of top 10 cryptocurrencies in billions USD'
```

In [5]:
```
market_caps
```

Out[5]:
```
0    954.7
1    514.4
2     95.8
3     76.3
4     57.9
5     45.7
6     41.0
7     38.7
8     28.8
9     25.8
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

we can see that the data itself is typed, i.e.:

In [6]:
```
market_caps.dtype
```

Out[6]:
```
dtype('float64')
```

and we can see that the values of the Series are just simple numpy arrays

In [7]:
```
type(market_caps.values)
```

Out[7]:
```
numpy.ndarray
```

One key difference between `np.ndarray` and `pd.Series` is that `pd.Series` is indexed. In the above example, the index happens to be a sequential index from 0 to 9, however we can easily change this index:

In [8]:
```
market_caps.index = ['BTC', 'ETH', 'BNB', 'USDT', 'SOL', 'ADA', 'USDC', 'XRP', 'DOT',
```

In [9]:
```
market_caps
```

Out[9]:
```
BTC     954.7
ETH     514.4
BNB      95.8
USDT     76.3
SOL      57.9
ADA      45.7
USDC     41.0
XRP      38.7
DOT      28.8
LUNA     25.8
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

It seems like a little bit of a pain in the ass to do this step by step, but we can do this all at once with `pd.Series` constructor for dicts:

In [10]:
```
pd.Series({
    'BTC': 954.7,
    'ETH': 514.4,
    'BNB': 95.8,
    'USDT': 76.3,
    'SOL': 57.9,
    'ADA': 45.7,
    'USDC': 41.0,
    'XRP': 38.7,
    'DOT': 28.8,
    'LUNA': 25.8
}, name='Market caps of top 10 cryptocurrencies in billions USD')
```

Out[10]:
```
BTC     954.7
ETH     514.4
BNB      95.8
USDT     76.3
SOL      57.9
ADA      45.7
USDC     41.0
XRP      38.7
DOT      28.8
LUNA     25.8
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

or using lists:

In [11]:
```
pd.Series(
    [954.7, 514.4, 95.8, 76.3, 57.9, 45.7, 41.0, 38.7, 28.8, 25.8],
    index=['BTC', 'ETH', 'BNB', 'USDT', 'SOL', 'ADA', 'USDC', 'XRP', 'DOT', 'LUNA'],
    name='Market caps of top 10 cryptocurrencies in billions USD'
)
```

Out[11]:
```
BTC     954.7
ETH     514.4
BNB      95.8
USDT     76.3
SOL      57.9
ADA      45.7
USDC     41.0
XRP      38.7
DOT      28.8
LUNA     25.8
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

tldr; Series construction is pretty intuitive and **very** flexible

Series Indexes

Indices are very important to understand for both `pd.DataFrame` and `pd.Series`, and we'll start looking at indices with Series here. When you access an item from the series, you can do it by **index** value or **location**. To access by index you can simple do:

In [12]:
```
market_caps['BTC']
```

Out[12]:
```
954.7
```

In [13]:
```
market_caps['USDC']
```

Out[13]:
```
41.0
```

---

note: one thing that many people mess up at the beginning is thinking that the `[*]` syntax is for positional access. This mistakes happens because the **default** index for both `pd.Series` and `pd.DataFrame` is a sequential index, so index access and location access is the same. **HOWEVER** this is absolutely not true once the list is change (e.g. it's sorted).

Here's an example:

In [14]:
```
series = pd.Series([10, 2, 3])
```

In [15]:
```
series[0] # we expect 10
```

Out[15]:
```
10
```

In [16]:
```
series[2] # we expect 3
```

Out[16]:
```
3
```

In [17]:
```
series.sort_values(inplace=True)
```

In [18]:
```
series
```

Out[18]:
```
1     2
2     3
0    10
dtype: int64
```

In [19]:
```
series[0]
```

Out[19]:
```
10
```

You can see above, when we access **index** `0` of the sorted series above, we still get 10, but this is now the 3rd item in the series. This is a very common error when working with `pandas`.

To properly access an element in a series **by position**, use `.iloc[*]`

In [20]:
```
series[0]
```

Out[20]:
```
10
```

In [21]:
```
series.iloc[0]
```

Out[21]:
```
2
```

Selecting multiple items from the Series can be done by passing in a list:

In [22]:
```
market_caps[['BTC', 'DOT']]
```

Out[22]:
```
BTC    954.7
DOT     28.8
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

This also works with `.iloc[*]`

In [23]:
```
market_caps.iloc[[1, 5]]
```

Out[23]:
```
ETH    514.4
ADA     45.7
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

Finally, slicing can be done with both index and positional references:

In [24]:
```
market_caps['BTC':'BNB']
```

Out[24]:
```
BTC    954.7
ETH    514.4
BNB     95.8
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

In [25]:
```
market_caps.iloc[0:2]
```

Out[25]:
```
BTC    954.7
ETH    514.4
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

note: slicing with indices is inclusive of the end value (i.e. BNB is included in the result above), however slicing using position excludes the end value, which is consistent with slicing a python list

Series Filtering

If we want to select a subset of a Series by a condition against its value, we can first create a boolean series and then select on that:

In [26]:
```
market_caps < 50
```

Out[26]:
```
BTC     False
ETH     False
BNB     False
USDT    False
SOL     False
ADA      True
USDC     True
XRP      True
DOT      True
LUNA     True
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: bool
```

In [27]:
```
market_caps[market_caps < 50]
```

Out[27]:
```
ADA     45.7
USDC    41.0
XRP     38.7
DOT     28.8
LUNA    25.8
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

We can also combine conditions easily, e.g.:

In [28]:
```
market_caps[(market_caps < 50) & (market_caps.index.str.len() == 4)]
```

Out[28]:
```
USDC    41.0
LUNA    25.8
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

note: you need parentheses around each condition if you want to join them with `&`, `|` or `~` because it has the same priority of operations as the comparison operators

Operations and Aggregations

Because `pd.Series` run numpy underneath, all operations and aggregations are vectorized under the hood.

For example, multiplying by a scalar multiplies every item in the series

In [29]:
```
market_caps * 1_000_000_000
```

Out[29]:
```
BTC     9.547000e+11
ETH     5.144000e+11
BNB     9.580000e+10
USDT    7.630000e+10
SOL     5.790000e+10
ADA     4.570000e+10
USDC    4.100000e+10
XRP     3.870000e+10
DOT     2.880000e+10
LUNA    2.580000e+10
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

Aggregations are also standard, e.g.:

In [30]:
```
market_caps.mean()
```

Out[30]:
```
187.91
```

In [31]:
```
market_caps.sum()
```

Out[31]:
```
1879.0999999999997
```

In [32]:
```
market_caps.std()
```

Out[32]:
```
306.96937397003563
```

In addition, all numpy functions work with Series, since all Series are just `np.ndarray` under the hood:

In [33]:
```
np.log(market_caps)
```

Out[33]:
```
BTC     6.861397
ETH     6.243001
BNB     4.562263
USDT    4.334673
SOL     4.058717
ADA     3.822098
USDC    3.713572
XRP     3.655840
DOT     3.360375
LUNA    3.250374
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

In [34]:
```
np.sum(market_caps)
```

Out[34]:
```
1879.0999999999997
```

Mutating the Series

While most Series functions in pandas will not modify the original Series (e.g. `*`, filtering and slicing will all create new `pd.Series`), we can modify elements of a series in place:

In [35]:
```
market_caps
```

Out[35]:
```
BTC     954.7
ETH     514.4
BNB      95.8
USDT     76.3
SOL      57.9
ADA      45.7
USDC     41.0
XRP      38.7
DOT      28.8
LUNA     25.8
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

In [36]:
```
market_caps['BTC'] = 1000
market_caps
```

Out[36]:
```
BTC     1000.0
ETH      514.4
BNB       95.8
USDT      76.3
SOL       57.9
ADA       45.7
USDC      41.0
XRP       38.7
DOT       28.8
LUNA      25.8
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

In [37]:
```
market_caps.iloc[3] = 0
market_caps
```

Out[37]:
```
BTC     1000.0
ETH      514.4
BNB       95.8
USDT       0.0
SOL       57.9
ADA       45.7
USDC      41.0
XRP       38.7
DOT       28.8
LUNA      25.8
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```

In [39]:
```
market_caps[market_caps < 50] = 0
market_caps
```

Out[39]:
```
BTC     1000.0
ETH      514.4
BNB       95.8
USDT       0.0
SOL       57.9
ADA        0.0
USDC       0.0
XRP        0.0
DOT        0.0
LUNA       0.0
Name: Market caps of top 10 cryptocurrencies in billions USD, dtype: float64
```